

# 000 001 002 003 004 005 006 007 008 009 010 011 012 013 014 015 016 017 018 019 020 021 022 023 024 025 026 027 028 029 030 031 032 033 034 035 036 037 038 039 040 041 042 043 044 045 046 047 048 049 050 051 052 053 QUEST: INCENTIVIZING LLMs TO GENERATE DIFFI- CULT PROBLEMS

Anonymous authors

Paper under double-blind review

## ABSTRACT

Large Language Models have achieved strong performance on reasoning tasks, solving competition-level coding and math problems. However, their scalability is limited by human-labeled datasets and the lack of large-scale, challenging coding problem training data. Existing competitive coding datasets contain only thousands to tens of thousands of problems. Previous synthetic data generation methods rely on either augmenting existing instruction datasets or selecting challenging problems from human-labeled data. In this paper, we propose QueST, a novel framework which combines difficulty-aware graph sampling for prompt and difficulty-aware rejection fine-tuning that directly optimizes specialized generators to create challenging coding problems. Our trained generators demonstrate superior capability at creating challenging problems compared to even proprietary models such as GPT-4o. We leverage this method to generate large-scale synthetic coding problems, which we then use to distill from long Chain-of-Thought (CoT) models or conduct reinforcement learning for smaller models, proving effective in both scenarios. Our distilled model achieves the best performance compared to similarly sized models trained on previous long CoT SFT datasets. By training generators to create more difficult problems, QueST pushes the boundaries of reasoning abilities in large language models.

## 1 INTRODUCTION

Test-time scaling through long chain-of-thought and large-scale reinforcement learning has dramatically boosted the reasoning ability of large language models, enabling LLMs to solve competition-level coding and math problems that were previously beyond their reach. Models like OpenAI o1 (OpenAI, 2024) and DeepSeek-R1 (Guo et al., 2025) have demonstrated remarkable performance on challenging benchmarks such as Codeforces, AIME, and IOI, achieving expert-level problem-solving capabilities through extensive reasoning traces that can span thousands of tokens. However, the problems used for training these models even require expert-level human annotations, which severely limits the scalability of LLM training. Current competitive coding datasets (Li et al., 2022; 2023) contain only thousands to tens of thousands of problems. As LLMs become more capable, the requirements for training data grow increasingly demanding—often requiring PhD-level experts in mathematics, computer science, and algorithm design to propose novel problems that can genuinely challenge these models. This process is not only extremely costly for requiring experts to propose difficult problems but also fundamentally cannot scale in terms of both dataset size and problem difficulty, creating a critical bottleneck in the development of next-generation reasoning models.

To mitigate this problem, methods for synthetic data generation and augmentation have been proposed. Previous works have focused either on paraphrasing-based augmentation (Luo et al., 2025a; Yu et al., 2024) or extracting concepts and recombining them based on co-occurrence probabilities (Tang et al., 2024; Zhao et al., 2025). Some recent works have proposed leveraging model weaknesses and extracting concepts to create new problems (Liang et al., 2025). More recently, reasoning-based LLMs have presented the next paradigm in advancing large language model reasoning capabilities (OpenAI, 2024; Guo et al., 2025). Works like Guo et al. (2025); Guha et al. (2025) created long chain-of-thought responses from reasoning models and curated synthetic SFT datasets, effectively helping small open-weight LLMs achieve superior performance in code and math tasks. Ahmad et al. (2025) curated the largest open-source dataset by obtaining long CoT responses from DeepSeek-R1 multiple times for each problems, though the problems themselves are still sourced from human-

054  
055  
056 Table 1: Comparison between representative code reasoning datasets.  
057  
058  
059  
060  
061  
062

| Code Datasets                          | #Problems   | Long CoT Responses | Synthetic Problems |
|--|-------------|--------------------|--------------------|
| CodeContest (Li et al., 2022)          | 13K         | ✗                  | ✗                  |
| TACO (Li et al., 2023)                 | 26K         | ✗                  | ✗                  |
| Bespoke-Stratos (Labs, 2025)           | 17K         | ✓                  | ✗                  |
| Open-R1 Codeforces-cots (Face, 2025)   | 10K         | ✓                  | ✗                  |
| OpenCodeReasoning (Ahmad et al., 2025) | 28K         | ✓                  | ✗                  |
| Ours (QueST)                           | <b>100K</b> | ✓                  | ✓                  |

063  
064 labeled competition coding problems. These methods have narrowed the performance gap between  
065 open-weight models and closed-weight reasoning models.  
066

067 However, existing methods described above either focus on leveraging existing human-annotated  
068 problems and curating synthetic responses from existing reasoning models, or rely on a *fixed* LLM to  
069 generate new problems by prompting. In this paper, we are the first to propose a method that directly  
070 trains an LLM generator to create challenging competitive code reasoning problems. We call our  
071 method **QueST**, embarking on a quest to generate increasingly challenging code problems through  
072 the combination of difficulty-aware graph sampling and difficulty-aware rejection fine-tuning. This  
073 approach is more scalable and flexible compared to previous methods that used a fixed generator or  
074 fixed human-labeled problems. Our proposed method makes the generator specialized and stronger  
075 than even closed-weight strong instruction models at creating challenging problems. We leverage this  
076 to generate the largest-scale code problem training set compared to previous synthetic data approaches,  
077 and the statistics of our synthetic data with previous data are shown in Table 1. We obtained responses  
078 from long chain-of-thought reasoning models, then leveraged the generated datasets to SFT small  
079 models, achieving competitive scores for similar-sized models on code reasoning benchmarks like  
080 LiveCodeBench (Jain et al., 2025) and USACO (Shi et al., 2024).  
081

Our contributions can be summarized as follows:

- 082 • We introduce a novel difficulty-aware coding problems generation framework that combines  
083 both difficulty-aware graph sampling and difficulty-aware rejection fine-tuning, which trains  
084 specialized generators to create challenging coding problems.
- 085 • We create the largest synthetic code reasoning problem set to date, comprising over 100K  
086 challenging coding problems paired with detailed chain-of-thought solutions from reasoning  
087 models.
- 088 • We demonstrate that small models fine-tuned on our synthetic dataset achieve competitive  
089 performance among similarly-sized models on LiveCodeBench and USACO benchmarks.  
090 We also demonstrate effectiveness in RL experiments.
- 091 • We conduct comprehensive ablation studies and analyses of our proposed method and the  
092 distribution of the generated coding problems.

## 094 2 QUEST

095 In this section, we present QUEST, our proposed method for generating difficult problems. We  
096 focus our investigation on the generation of coding problems, as other forms of reasoning tasks  
097 (e.g., mathematical reasoning) can be regarded as special cases of coding tasks (Jiao et al., 2025).  
098 We begin by introducing our scaffolding framework for problem generation, which builds upon  
099 MathScale (Tang et al., 2024). Next, we detail our strategies for incentivizing LLMs to produce more  
100 difficult problems. Finally, we demonstrate how our scaffolding can be adapted to further enhance  
101 the generation of challenging problems.  
102

### 104 2.1 PRELIMINARY: PROBLEM GENERATION THROUGH CONCEPT GRAPH

105 Our scaffolding for problem generation is based on (Tang et al., 2024), which generates new problems  
106 based on existing seed problems by prompting an LLM in three steps (i.e., concept extraction, graph  
107 construction and problem generation).

108 **Concept Extraction** For each problem  $q$  in the seed problem set  $\mathbf{Q}_{\text{seed}}$ , we prompt an LLM to  
 109 extract concepts  $c$  (topics and knowledge points) from it. We follow the setting of Tang et al. (2024),  
 110 topics refers to general directions, knowledge points refers to more fine-grained concepts, example  
 111 can be found in Appendix Table 5. Note that problem generation can be guided later using the  
 112 concepts that we extracted in this step. The process can be defined formally as follows:

$$113 \quad \mathbf{C} = \mathcal{G}^Q((p_{\text{extract}}, \mathbf{Q}_{\text{seed}})) \quad (1)$$

115 where  $p_{\text{extract}}$  is the prompt used to extract concepts,  $\mathcal{G}^Q$  is our problem generator and  $\mathbf{C}$  is the set of  
 116 concepts extracted. Detail prompts of  $p_{\text{extract}}$  are in Appendix Table 7.

117 **Graph Construction** Once we obtain the concepts  $\mathbf{C}$ , we proceed to identify plausible combi-  
 118 nations of these concepts. Two concepts are considered to form a reasonable combination if they have  
 119 frequently co-occurred within the same problem in the seed dataset. To capture these relationships,  
 120 we construct a concept graph in which nodes represent individual concepts, and edge weights encode  
 121 the strength of co-occurrence between concept pairs. The edge weight  $w(u, v)$  is defined as follows:

$$123 \quad w(u, v) = \log(\text{freq}(u, v) + \epsilon) \quad (2)$$

124 where  $u$  and  $v$  denote concept nodes, and  $\text{freq}(u, v)$  represents the observed co-occurrence frequency  
 125 of these concepts. A small constant  $\epsilon$  is added to ensure numerical stability by preventing zero counts.

127 Given the constructed graph, we proceed to sample concept combinations, which are then utilized for  
 128 the generation of new problems. We start from a uniformly random sampling from all the topics and  
 129 subsequently perform up to six steps of a random walk (Tang et al., 2024). At each step,  
 130 the transition probability from node  $u$  to node  $v$  is defined as:

$$131 \quad p_{u,v} = \frac{\exp(w(u, v))}{\sum_{v' \in \mathcal{N}(u)} \exp(w(u, v'))} \quad (3)$$

133 where  $\mathcal{N}(u)$  denotes the set of nodes adjacent to  $u$ . After each random walk episode, we obtain a  
 134 sampled concept combination  $s$ , which is subsequently used for problem generation.

136 **Problem Generation** Given the sampled concept set  $s$ , we leverage an LLM to generate new  
 137 problems. We incorporate few-shot examples to guide the LLM in formulating problems. These  
 138 examples are selected from the pool of seed problems based on the Jaccard distance between their  
 139 respective sets of concepts. Formally, this process can be described as:

$$140 \quad \mathbf{Q}_{\text{new}} = \mathcal{G}^Q(p_{\text{generate}}, \mathcal{S}(\mathbf{C}), \mathbf{Q}_{\text{seed}}) \quad (4)$$

142 where  $\mathcal{S}(\mathbf{C})$  denotes the set of sampled concepts, and  $p_{\text{generate}}$  represents the prompt template utilized  
 143 for problem generation (see Appendix Table 5 for additional details).

144 At this stage, our problem generator is designed to produce new problems, rather than explicitly  
 145 targeting increased difficulty. The generation of more challenging problems will be addressed in the  
 146 subsequent sections.

## 148 2.2 DIFFICULTY-AWARE REJECTION FINETUNING

150 We focus primarily on the generation of challenging coding problems, though our approach is readily  
 151 extensible to other forms of reasoning tasks. We first present our method for measuring problem  
 152 difficulty, and then illustrate how this measure is employed to guide LLMs in producing more difficult  
 153 problems.

154 **Difficulty Estimation** A natural way to assess the difficulty of a generated problem is to examine  
 155 the consistency of the models multiple outputs. Wang et al. (2023a) finds that self-consistency is  
 156 highly correlated with accuracy, which reflects the uncertainty of the models, and also the difficulty of  
 157 the problem. When most solutions converge to a single outcome, the problem is likely straightforward.  
 158 Conversely, if the solutions diverge and produce inconsistent outputs, this indicates model uncertainty,  
 159 suggesting the problem is more difficult.

161 Building on this intuition, we define the difficulty of a problem using the *average majority voting  
 rate* of its solutions. We illustrate our metric using coding problems as a case study, noting that

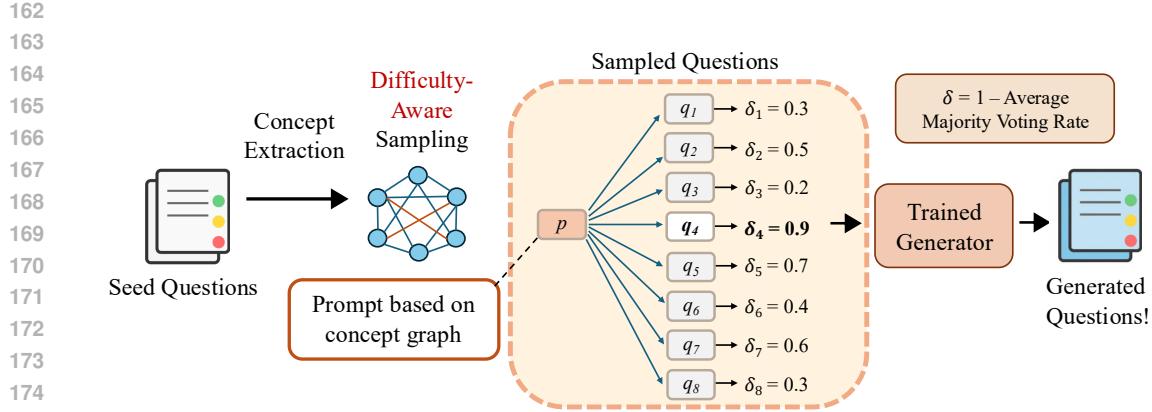


Figure 1: The pipeline of QueST. We first extract concepts based on seed problems, then use difficulty-aware sampling method described in Equation 10 to create prompts for problem generation. We generate 8 problems for each prompt, calculate the difficulty  $\delta$  of the generated problem based on Equation 6, and use the most difficult problem as rejection fine-tuning data to train our generator.

other verifiable reasoning problems (e.g. math) can be regarded as a special case of this setting. Let  $q \sim \mathcal{G}^Q(p_{\text{generate}}, s, \mathbf{Q}_{\text{seed}})$  denote a generated coding problem (see Equation (4)), where  $s \sim \mathcal{S}(\mathbf{C})$  is a sampled concept combination (see Section 2.1). The estimation proceeds in three steps. First, we prompt  $\text{gpt-4o}$  to generate  $T$  test inputs, forming the set  $\mathcal{I} = \{i_1, i_2, \dots, i_T\}$  (details of the prompt in Appendix Table 6). Second, we obtain  $M$  candidate solutions  $\mathcal{Y} = \{y_1, y_2, \dots, y_M\}$  from  $\text{gpt-4o}$ . Third, we execute each  $y_m \in \mathcal{Y}$  on all inputs  $i_t \in \mathcal{I}$ , producing output sets  $\mathcal{O}_t = \{g(y_1, i_t), g(y_2, i_t), \dots, g(y_M, i_t)\}$ , where  $g(y_m, i_t)$  denotes extracting the code from  $y_m$ , running it on input  $i_t$ , and recording the output. For each test input, the most likely output  $o_t$  is identified as the most frequent element in  $\mathcal{O}_t$ :

$$o_t = \arg \max_{o \in \mathcal{O}_t} f(o, \mathcal{O}_t) \quad (5)$$

where  $f(o, \mathcal{O}_t) = |\{x \in \mathcal{O}_t \mid x = o\}|$  counts the occurrences of  $o$  in  $\mathcal{O}_t$ . Finally, we quantify the problem difficulty as

$$\delta(q) = 1 - \frac{1}{T} \sum_{t=1}^T \frac{f(o_t, \mathcal{O}_t)}{M} \quad (6)$$

Intuitively,  $\delta(q)$  measures the degree of disagreement among candidate solutions: the lower the majority voting rate, the higher the difficulty. Thus, larger values of  $\delta(q)$  correspond to more challenging problems.

To further enhance the probability of generating valid synthetic problems, we filter out problems where over half of the test case outputs from generated responses return `None`, indicating unsuccessful code execution.

**Rejection Fine-tuning** Having introduced the difficulty measure  $\delta(q)$ , we now describe how it is employed to construct a dataset of prompt–problem pairs for training LLMs to generate difficult problems. The key idea is to sample multiple candidate problems from the same prompt and retain only the most difficult one.

As discussed in Section 2.1, for each concept combination  $s$ , a problem can be generated via

$$q \sim \mathcal{G}^Q(p_{\text{generate}}, s, \mathbf{Q}_{\text{seed}}) \quad (7)$$

where  $\mathcal{G}^Q$  denotes the LLM-based generator. More generally, let  $M_\theta$  be the LLM parameterized by  $\theta$ , and let  $p$  denote the *actual* prompt ( $p_{\text{generate}}$  instantiated with concept set  $s$  and seed problems  $\mathbf{Q}_{\text{seed}}$ ) used to query  $M_\theta$ . By sampling  $K$  times, we obtain a set of candidate problems:

$$q_k \sim M_\theta(p) \quad \text{for } k = 1, \dots, K \quad (8)$$

216 We denote this set by  $\mathcal{Q} = \{q_1, q_2, \dots, q_K\}$ . We then select the most difficult problem according to  
 217 our measure  $\delta(\cdot)$ :

$$218 \quad q^* = \arg \max_{q_k \in \mathcal{Q}} \delta(q_k) \quad (9)$$

220 Only  $q^*$  is retained, while the remaining candidates are discarded. The resulting pair  $(p, q^*)$  is added  
 221 to the training set  $\mathcal{D}_{\text{hard}}$ , which is used to fine-tune the problem generator  $M_\theta$ .

### 222 2.3 DIFFICULTY-AWARE GRAPH CONSTRUCTION

224 This section extends our problem generation scaffolding (Section 2.1) to be difficulty-aware. In  
 225 the baseline setup, the initial edge weights of the concept graph are determined primarily by the  
 226 co-occurrence statistics of concepts within the same problems. Here, we further incorporate difficulty  
 227 by modeling the hardness of concepts with respect to the difficulty levels of the problems in which  
 228 they appear. Since each problem in the seed dataset (e.g., TACO; Li et al. (2023)) is annotated with  
 229 human-curated difficulty labels, we leverage this information when constructing the concept graph  
 230 for problem generation prompts. Specifically, beyond using co-occurrence counts as edge weights for  
 231 random walk sampling, we also incorporate the average difficulty of all problems that involve both  
 232 concepts connected by an edge. The new edge weights are defined as

$$233 \quad w(u, v) = \log(\alpha \cdot \text{freq}(u, v) + (1 - \alpha) \cdot \text{diff}(u, v) + \epsilon) \\ 234 \quad \text{where } \text{diff}(u, v) = \frac{1}{|Q_{u,v}|} \sum_{q \in Q_{u,v}} d(q), \quad Q_{u,v} = \{q \mid u \in q, v \in q\}. \quad (10)$$

237 Here,  $\alpha$  is a hyperparameter that balances the contribution of co-occurrence frequency and difficulty  
 238 and we set  $\alpha = 0.2$  in our experiments. The constant  $\epsilon$  is included to avoid taking the logarithm  
 239 of zero. The set  $Q_{u,v}$  consists of all problems containing both concepts  $u$  and  $v$ ; its cardinality is  
 240 denoted by  $|Q_{u,v}|$ . Finally,  $d(q)$  represents the human-annotated difficulty level of problem  $q$ , given  
 241 as an integer from 1 to 5.

## 242 3 EXPERIMENTS

244 In this section, we present the detail of our experiments. We first use our proposed difficulty measure  
 245 method for data selection. Then we show the long CoT SFT results using datasets distilled from  
 246 Qwen3-8B compared with previous strong baselines, and we show our generated datasets can also be  
 247 effective when used in RL training. We further present an ablation study to investigate the effect of  
 248 each role in our proposed method. Finally we have contamination analysis and statistics about our  
 249 generated data.

### 251 3.1 IMPLEMENTATION DETAILS

253 **Seed data:** We use TACO (Li et al., 2023) as seed data, which has human-annotated labels for  
 254 difficult. TACO has 25.4K training samples and 1K test samples. Each problems is annotated with  
 255 difficulty, test cases, and a list of topics. Samples in this dataset are collected from open-access sites  
 256 where programmers share problems with each other, including Aizu, AtCoder, CodeChef, Codeforces,  
 257 and LeetCode.

258 **Benchmarks** We use LiveCodeBench-V5 (Jain et al., 2025) and USACO as our evaluation benchmarks.  
 259 We use LiveCodeBench-V5 for direct comparison with a strong baseline (Ahmad et al., 2025);  
 260 USACO (Shi et al., 2024) is used because it is a representative code competition which contains  
 261 difficult problems and has already been curated as benchmark for evaluation.

262 **Models:** We use Qwen3-8B as our teacher model in distillation experiments, as it is efficient and has  
 263 competitive reasoning performance. We use Qwen2.5-Coder-7B-Instruct and Qwen3-8B-Base as our  
 264 student model, respectively. For the RL experiments, we use Qwen2.5-7B-Instruct model as starting  
 265 checkpoint for small-scale verification. We use Qwen2.5-14B-Instruct and GPT-4o as generators, as  
 266 they can follow instructions relatively well compared to smaller models.

267 **Hyperparameters:** We use vLLM<sup>1</sup> as our inference framework for both distillation and evaluation  
 268 experiments. We set temperature to 0.6 for all experiments. We set the batch size to 128 and the

269 <sup>1</sup><https://github.com/vllm-project/vllm>

270  
271  
272  
273  
274 Table 2: Effect of different strata of synthetically generated coding problems on downstream perfor-  
275 mance.  $\delta$  refers our estimated difficulty defined in Section 2.2 . Response length is determined based  
276 on responses generated by Qwen3-8B.  
277  
278  
279

| Selection of problems | LiveCodeBench-V5 score | Avg. response length in tokens |
|-----------------------|------------------------|--------------------------------|
| Random 3K             | 36.29                  | 11.9K                          |
| Highest $\delta$ 3K   | 39.28                  | 14.2K                          |
| Median $\delta$ 3K    | 36.35                  | 14.1K                          |
| Lowest $\delta$ 3K    | 32.37                  | 6.8K                           |
| Longest response 3K   | 38.35                  | 22.6K                          |

280  
281  
282 learning rate to 5e-5 for our SFT experiments, including the fine-tuning of the generator models. We  
283 use VeRL<sup>2</sup> for our RL experiments, and use 128 as the rollout batch size, 64 as the mini-batch size,  
284 and 16 as the rollout sample size. For all evaluation, we calculate averaged pass@1 across 16 runs.  
285

### 286 3.2 USING ESTIMATED DIFFICULTY FOR DATA SELECTION

287 Before training the generator to produce difficult coding problems, we first need a trustworthy signal  
288 that can serve as a proxy for difficulty when gold labels are unavailable for generated problems. As  
289 mentioned above, we propose using  $\delta$  we defined in Section 2.2 based on model responses. To verify  
290 the usefulness of this signal, we conduct a preliminary experiment that selects subsets of generated  
291 problems based on this signal for controlled comparison. We use our baseline graph random walking  
292 process to generate 50K problems using TACO as seed data. For each problem, we generate 8  
293 responses and compute  $\delta$ . We then select 3K samples with the highest  $\delta$ , 3K with the lowest  $\delta$ , 3K  
294 with  $\delta$  closest to 0.5, and an additional 3K randomly sampled for comparison. We also use response  
295 token length as another difficulty proxy and select 3K samples with the longest responses. Table 2  
296 shows the results of using different selection methods and the performance of models trained on the  
297 selected problems, with 8 responses generated for each problem to ensure the scale and significance  
298 of our experiments. We observe that problems with the highest  $\delta$  achieve the best performance, even  
299 surpassing those with the longest token responses, and using significant less tokens. We can also  
300 observe that for the problems with highest  $\delta$ , the token length is higher than problems with median  
301 and lowest  $\delta$ , which indicates there are some positive correlations between token length and  $\delta$ , but  $\delta$   
302 is still a more effective and efficient signal compared to response length.  
303

### 304 3.3 TRAINED GENERATOR FOR DISTILLATION

305 We then use our trained generator to generate problems and leverage these problems to obtain  
306 responses from long chain-of-thought models (Qwen3-8B in our experiments) for training student  
307 models. In Table 3, we conduct a comprehensive comparison between previous Long CoT SFT  
308 datasets and our generated datasets on representative code reasoning benchmarks: LiveCodeBench-  
309 V5 and USACO. For our method (QueST), as described in Section 3.1, we use Qwen2.5-14B-Instruct  
310 to train a specialized generator under our reject fine-tuning and difficulty aware graph sampled  
311 prompt, 20K and 100K represents the training data size. "7B" means we trained from Qwen2.5-  
312 Coder-7B-Instruct, and "8B" means we trained from Qwen3-8B-Base.  
313

314 Among the comparison group, OCR-Qwen-7B-Instruct Ahmad et al. (2025) stands out as the strongest  
315 competitor, leveraging DeepSeek-R1 as the teacher model and generating up to 32 responses for each  
316 of the 28K humanwritten coding problems. To ensure a fair comparison, we re-implement the OCR  
317 method using Qwen3-8B-Base as the student model and generate 4 responses per problem (yielding  
318 a total of 112K examples) using Qwen3-235B-A22B as the teacher. Even under these conditions,  
319 QueST-100K-8B outperforms OCR-8B across both benchmarks.  
320

### 321 3.4 REINFORCEMENT LEARNING

322 Our generated data can also be used for RLVR (Reinforcement Learning with Verifiable Reward).  
323 We use majority voting results produced by Qwen3-8B as pseudo output labels for each test case

<sup>2</sup><https://github.com/volcengine/verl>

324  
 325 Table 3: Performance on LiveCodeBench-V5 and USACO. Note: In our method, we only use  
 326 Qwen3-8B as teacher model for 7B model, and Qwen3-235B-A22B as teacher model for 8B model.  
 327 The content in brackets represents the generator models used for problem generation (GPT-4o for  
 328 MathScale), in our methods, we use our trained Qwen2.5-14B-Instruct as generator. 20K and 100K  
 329 means the number of training samples.

| Model                               | LiveCodeBench-V5 |             |             |             | USACO       |             |             |             |
|-------------------------------------|------------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|
|                                     | Easy             | Medium      | Hard        | Avg.        | Easy        | Medium      | Hard        | Avg.        |
| <b>Upper Bound (Teacher Models)</b> |                  |             |             |             |             |             |             |             |
| DeepSeek-R1                         | 98.5             | 79.8        | 37.4        | 65.6        | -           | -           | -           | -           |
| Qwen3-8B                            | 94.0             | 74.1        | 28.9        | 58.7        | 58.5        | 42.8        | 22.3        | 43.5        |
| <b>Baselines</b>                    |                  |             |             |             |             |             |             |             |
| OpenThinker-7B (114K)               | 80.6             | 16.9        | 1.6         | 25.5        | 11.0        | 2.1         | 0.0         | 5.0         |
| R1-Distill-Qwen-7B (800K)           | 86.6             | 43.8        | 7.0         | 38.0        | 22.9        | 9.7         | 3.8         | 13.4        |
| OlympicCoder-7B (100K)              | 82.1             | 49.4        | 12.2        | 40.9        | 31.4        | 12.5        | 1.3         | 17.0        |
| OCR-Qwen-7B-Instruct (700K)         | 95.4             | 64.0        | 18.0        | 51.3        | 41.5        | 26.0        | 7.5         | 27.2        |
| MathScale-20K-7B (GPT-4o)           | 82.8             | 36.6        | 8.2         | 34.9        | 28.0        | 15.6        | 1.3         | 16.7        |
| <b>Our Method</b>                   |                  |             |             |             |             |             |             |             |
| QueST-20K-7B                        | 84.9             | 41.4        | 10.4        | 37.9        | 30.5        | 16.7        | 6.2         | 19.4        |
| QueST-100K-7B                       | 87.8             | 50.8        | 14.6        | 43.3        | 31.4        | 25.0        | 10.0        | 23.5        |
| OCR-8B (112K, Our Impl.)            | 96.0             | 70.2        | 26.2        | 56.5        | 54.5        | 40.5        | 22.9        | 41.3        |
| QueST-100K-8B (100K, Ours)          | 97.1             | 74.8        | 28.4        | 59.4        | 55.9        | 44.0        | 24.7        | 43.5        |
| QueST-8B (212K, Ours)               | <b>97.6</b>      | <b>81.0</b> | <b>36.6</b> | <b>65.2</b> | <b>65.5</b> | <b>48.6</b> | <b>28.7</b> | <b>49.9</b> |

348  
 349 Table 4: RL results on LiveCodeBench-V5  
 350

| Model                           | LiveCodeBench-V5 |        |      |      |
|---------------------------------|------------------|--------|------|------|
|                                 | Easy             | Medium | Hard | Avg. |
| Qwen2.5-7B-Instruct             | 47.4             | 8.4    | 0.1  | 14.3 |
| Qwen2.5-7B-Instruct TACO RL     | 56.7             | 10.8   | 1.1  | 17.3 |
| Qwen2.5-7B-Instruct Baseline RL | 56.0             | 9.6    | 3.2  | 17.6 |
| Qwen2.5-7B-Instruct QueST RL    | 56.4             | 9.6    | 4.8  | 18.6 |

358  
 359 of each generated problem. Since our generated test cases are not guaranteed to be valid, we filter  
 360 out test cases where over half of the outputs are none (indicating failed execution for generated  
 361 solutions), then keep the remainder for RLVR. We use the GRPO (Shao et al., 2024) algorithm to train  
 362 Qwen2.5-7B-Instruct on 12K problems sampled from TACO, 6K data from our baseline synthetic  
 363 method (mathscale) (Tang et al., 2024), and 6K data from QueST. We report our results in Table 4,  
 364 which shows effectiveness of our proposed method.

365  
 366 We report the training reward curve during the training process in Figure 2. It shows that the model  
 367 trained on TACO datasets gains the highest reward score during the whole training stage, our baseline  
 368 synthetic method gains a lower score, and the model trained on a dataset generated by the QueST  
 369 method gains the lowest score. The training reward can serve as a proxy of the inverse difficulty of  
 370 these three different datasets.

### 371 3.5 ABLATION STUDY

372  
 373 We conducted an ablation study for fair comparison across different settings, as shown in Table  
 374 5. In the first two rows of the table, we examine whether using difficulty-aware graph random  
 375 walking improves performance when using GPT-4o as the generator. The results demonstrate that the  
 376 difficulty-aware graph achieves clear improvement. In the third, fourth and fifth rows, we compare  
 377 performance when only using Qwen2.5-14B-Instruct (Baseline), with the difficulty-aware graph  
 378 but Qwen2.5-14B-Instruct without further training and Qwen2.5-14B-Instruct under our rejection  
 379 fine-tuning method (QueST). The results show that when using difficulty-aware random sampling

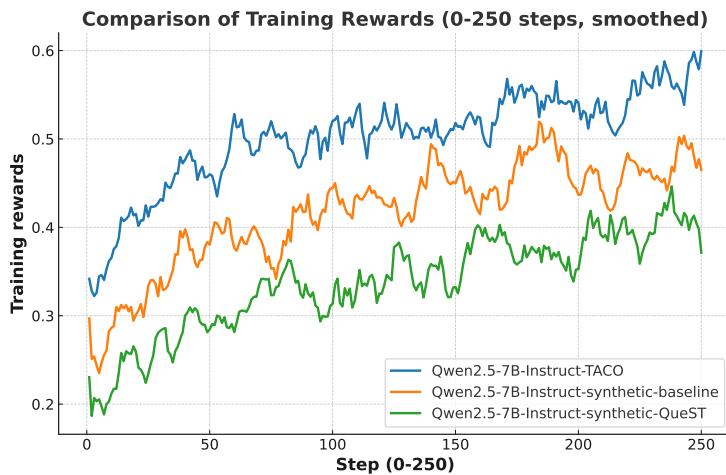


Figure 2: Training rewards comparison in the training process of RL under different datasets.

prompts, our fine-tuned generator can bring better performance than the model without using our fine-tuning method, and better than baseline. Therefore, Table 5 indicate both difficulty-aware sampling and rejection fine-tuning have positive effect and lead to generating difficult problems.

Table 5: Ablation study on LiveCodeBenchV5. “Baseline” here represents the our baseline problem generation pipeline (Tang et al., 2024) which we discussed in Section 2.1. Here we generate 20K questions for all settings to fair comparison, and the base model we used to train is Qwen2.5-Coder-7B-Instruct. “RFT” is abbreviation of our rejection fine-tuning method.

| Methods   | LiveCodeBench |        |      |      |
|---|---------------|--------|------|------|
|   | Easy          | Medium | Hard | Avg. |
| Problem Generator: GPT-4o                         |               |        |      |      |
| Baseline  | 82.8          | 36.6   | 8.2  | 34.9 |
| Baseline w/ difficulty-aware graph                | 83.6          | 41.1   | 10.9 | 37.5 |
| Problem Generator Qwen2.5-14B-Instruct            |               |        |      |      |
| Baseline  | 81.7          | 32.6   | 6.1  | 32.8 |
| Baseline w/ difficulty-aware graph                | 85.0          | 39.2   | 8.0  | 36.1 |
| Baseline w/ difficulty-aware graph w/ RFT (QueST) | 84.9          | 41.4   | 10.4 | 37.9 |

### 3.6 ADDITIONAL ANALYSIS

We visualize and compare the 25 most sampled knowledge points with and without difficulty-aware sampling in Appendix Figure 4. The figure shows that knowledge points sampled more frequently by naive sampling than by difficulty-aware sampling tend to be more common overall, while knowledge points sampled less frequently by naive sampling tend to be less common. In other words, difficulty-aware sampling upweights infrequent knowledge points and downweights frequent knowledge points compared to naive sampling. The infrequent knowledge points are visualized in the left figure and are generally more difficult, including topics such as the “knapsack problem”, “Optimal Play Strategies”, and “prime factorization”, compared to the basic concepts shown in the right figure.

We also conduct a case study on generated problems from both original model and model trained by QueST framework in Appendix Table 6. It shows that the problem generated by our trained model requires more complex operations and more knowledge compared the question generated by original model.

432 We conduct contamination detection experiments on our generated datasets to exclude the effects  
 433 of data contamination on benchmark performance. Specifically, we compute token-based 50-gram  
 434 Jaccard similarity scores and the scores across all datasets and benchmarks we used are 0 which  
 435 indicates there is no contamination in our generated data.

436 We also conducted correlation analysis of our computed difficulty estimation with human labeled  
 437 difficulty ratings in TACO dataset. We transform difficulty levels of TACO to 1-5 scores as rating  
 438 levels, corresponding to easy, medium, medium hard, hard, and very hard, respectively. We report  
 439 the distribution and mean scores of average majority voting rate for each problems in each human  
 440 labeled rating in Figure 3. And it shows a trend that when rating become larger (more difficult), the  
 441 mean of average majority voting rate become lower.

## 4 RELATED WORK

### 4.1 SYNTHETIC DATA FOR LANGUAGE MODELS

447 Synthetic data has been widely used in training language models. Previous works have mainly  
 448 focused on using small sets of seed data and leveraging LLMs to augment them and generate larger  
 449 datasets. Some works (Honovich et al., 2023; Li et al., 2024a; Toshniwal et al., 2025; Wang et al.,  
 450 2023b; Tang et al., 2024) focus on sampling seed data as in-context learning exemplars to generate  
 451 new ones. Ge et al. (2025) proposed using personas to augment previous in-context learning synthetic  
 452 data generation methods. Xu et al. (2024); Luo et al. (2025a); Hu et al. (2025) focus on augmenting  
 453 existing samples to create more complex ones. Some methods have also explored how to generate  
 454 synthetic data from scratch (Li et al., 2024b; Xu et al., 2025). More recently, Qin et al. (2025)  
 455 investigated whether synthetic data follows similar scaling laws as real data. Shah et al. (2025); Kaur  
 456 et al. (2025); Didolkar et al. (2024) extract skills and generate new problems. PromptCoT (Zhao  
 457 et al., 2025) also generates challenging problems based on mathematical concepts and rationale.  
 458 Tong et al. (2024) also proposed a difficulty-aware method but focuses on synthetic responses for  
 459 challenging problems. Liang et al. (2025) extract concepts from failure cases and synthesize new  
 460 problems during RL training. Additionally, there is research focused on leveraging pretraining or  
 461 web data to generate reasoning data in general domains (Yuan et al., 2025; Yue et al., 2024). Our  
 462 QueST framework focuses on a new perspective that aims to train a difficulty-aware generator to  
 463 generate difficult problems.

### 4.2 CODE REASONING

465 Code reasoning is an important capability of large language models. The reasoning ability of language  
 466 models can be enhanced using chain-of-thought (Wei et al., 2022), RLLVR (OpenAI, 2024; Guo et al.,  
 467 2025; Lambert et al., 2025), and self-consistency (Wang et al., 2023a), in math (Hendrycks et al.,  
 468 2021) and code (Jain et al., 2025; Shi et al., 2024) domains. Muennighoff et al. (2025) and Ye  
 469 et al. (2025) focus on manually curating small-scale reasoning problems, which is sufficient to boost  
 470 models’ reasoning ability. More recently, Face (2025), Ahmad et al. (2025), and Guha et al. (2025)  
 471 have developed large-scale distillation methods from reasoning models to obtain high-quality long  
 472 CoT SFT datasets that can be used to train student models effectively. Nvidia et al. (2024) curate  
 473 reasoning datasets throughout the entire training pipeline. Li et al. (2025) introduce an innovative  
 474 paradigm that transforms traditional code reasoning tasks from their original format into a “given  
 475 code + test cases / input-output prediction” structure. Complementing these supervised learning  
 476 approaches, Luo et al. (2025b) demonstrate the effectiveness of reinforcement learning techniques  
 477 applied to verified code reasoning problems. However, how to generate difficult synthetic coding  
 478 problems and use them for training remains relatively underexplored.

## 5 CONCLUSION

481 In this paper, we propose a method for generating difficult code problems at scale. Specifically, we  
 482 investigated a pipeline that uses majority voting to compute a proxy of difficulty and employs this as a  
 483 signal for rejection fine-tuning of the problem generator, and combined it with novel difficulty-aware  
 484 graph sampling prompts. This enables the trained generator to produce challenging problems at scale.  
 485 We then use these generated problems for supervised fine-tuning (SFT) and reinforcement learning  
 (RL) to verify their effectiveness. As a novel synthetic data generation method, we compared our

approach with previous baselines at similar scales on code reasoning benchmarks and show that our method achieves better performance even when using less SFT data, particularly for hard problems.

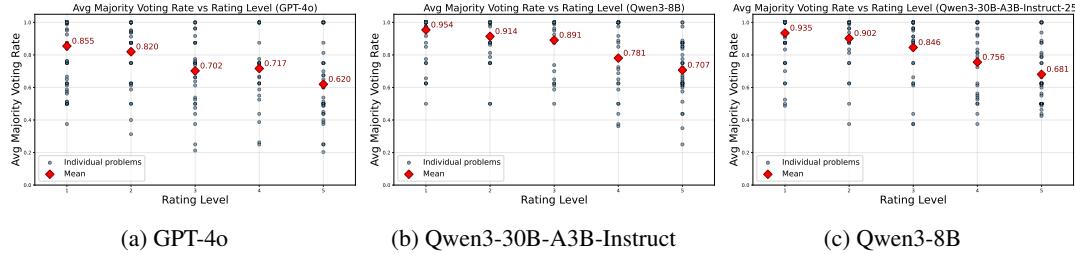


Figure 3: Avg Majority Voting Rate vs Rating Level (Difficulty) in TACO.

## LIMITATIONS AND FUTURE WORK

Although our method shows promise for rejection fine-tuning a generator, we still face limitations as the generator hasn't been trained using RL. One primary reason is that our current difficulty calculation is computationally expensive and challenging to implement in real-time to provide difficulty rewards in an RL pipeline, considering that we need to generate 8 responses and 20 test cases for each problem on the fly, execute them, and generate  $K$  problems for each prompt. In future work, it would be worthwhile to explore methods that can provide rewards in real time, such as directly training a reward model to predict difficulty, or investigating other efficient approaches.

510  
511  
512  
513  
514  
515  
516  
517  
518  
519  
520  
521  
522  
523  
524  
525  
526  
527  
528  
529  
530  
531  
532  
533  
534  
535  
536  
537  
538  
539

540  
541

## REPRODUCIBILITY STATEMENT

542

To help community reproduce our work, we described details of implementation in Section 3.1, which reports the details of data, benchmark, models, and hyperparameters we use in our experiments. We also report the framework we use for training and inference. In Appendix Figure 5 6 7, we report the prompt template we use.

543

544

## ETHICS STATEMENT

545

546

In the paper, all the data we use is open-sourced. TACO (Li et al., 2023) has Apache-2.0 license. LiveCodeBench (Jain et al., 2025) and USACO (Shi et al., 2024) are collected from open part of common competition websites.

547

548

549

550

551

552

553

554

555

556

557

558

559

560

561

562

563

564

565

566

567

568

569

570

571

572

573

574

575

576

577

578

579

580

581

582

583

584

585

586

587

588

589

590

591

592

593

594 REFERENCES  
595

596 Wasi Uddin Ahmad, Sean Narenthiran, Somshubra Majumdar, Aleksander Ficek, Siddhartha Jain,  
597 Jocelyn Huang, Vahid Noroozi, and Boris Ginsburg. Opencodereasoning: Advancing data dis-  
598 tillation for competitive coding. In *Second Conference on Language Modeling*, 2025. URL  
599 <https://openreview.net/forum?id=aykM7KUVJZ>.

600 Aniket Rajiv Didolkar, Anirudh Goyal, Nan Rosemary Ke, Siyuan Guo, Michal Valko, Timothy P  
601 Lillicrap, Danilo Jimenez Rezende, Yoshua Bengio, Michael Curtis Mozer, and Sanjeev Arora.  
602 Metacognitive capabilities of LLMs: An exploration in mathematical problem solving. In *The*  
603 *Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL <https://openreview.net/forum?id=D19UyP4HYk>.

604 Hugging Face. Open R1: A fully open reproduction of DeepSeek-R1, January 2025. URL <https://github.com/huggingface/open-r1>.

605 Tao Ge, Xin Chan, Xiaoyang Wang, Dian Yu, Haitao Mi, and Dong Yu. Scaling synthetic data creation  
606 with 1,000,000,000 personas, 2025. URL <https://arxiv.org/abs/2406.20094>.

607 Etash Guha, Ryan Marten, Sedrick Keh, Negin Raoof, Georgios Smyrnis, Hritik Bansal, Marianna  
608 Nezhurina, Jean Mercat, Trung Vu, Zayne Sprague, Ashima Suvarna, Benjamin Feuer, Liangyu  
609 Chen, Zaid Khan, Eric Frankel, Sachin Grover, Caroline Choi, Niklas Muenennighoff, Shiye Su,  
610 Wanja Zhao, John Yang, Shreyas Pimpalgaonkar, Kartik Sharma, Charlie Cheng-Jie Ji, Yichuan  
611 Deng, Sarah Pratt, Vivek Ramanujan, Jon Saad-Falcon, Jeffrey Li, Achal Dave, Alon Albalak,  
612 Kushal Arora, Blake Wulfe, Chinmay Hegde, Greg Durrett, Sewoong Oh, Mohit Bansal, Saadia  
613 Gabriel, Aditya Grover, Kai-Wei Chang, Vaishaal Shankar, Aaron Gokaslan, Mike A. Merrill,  
614 Tatsunori Hashimoto, Yejin Choi, Jenia Jitsev, Reinhard Heckel, Maheswaran Sathiamoorthy,  
615 Alexandros G. Dimakis, and Ludwig Schmidt. Openthoughts: Data recipes for reasoning models,  
616 2025. URL <https://arxiv.org/abs/2506.04178>.

617 Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu  
618 Zhang, Shirong Ma, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou,  
619 Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei  
620 Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chong Ruan, Damai Dai, Deli Chen, Dongjie  
621 Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li,  
622 H. Zhang, Hanwei Xu, Honghui Ding, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li,  
623 Jingchang Chen, Jingyang Yuan, Jinhao Tu, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang,  
624 Jin Chen, Kai Dong, Kai Hu, Kaichao You, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean  
625 Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan  
626 Zhang, Minghua Zhang, Minghui Tang, Mingxu Zhou, Meng Li, Miaojun Wang, Mingming  
627 Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi  
628 Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu,  
629 Shangyan Zhou, Shanhua Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou,  
630 Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Tao Yun, Tian Pei, Tianyu Sun, T. Wang,  
631 Wangding Zeng, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao,  
632 Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin  
633 Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin,  
634 Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu  
635 Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao  
636 Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi  
637 Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan  
638 Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu,  
639 Yuyang Zhou, Y. X. Zhu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying  
640 Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie,  
641 Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun  
642 Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang,  
643 and Zhen Zhang. DeepSeek-R1 incentivizes reasoning in llms through reinforcement learning.  
644 *Nature*, 645(8081):633–638, Sep 2025. ISSN 1476-4687. doi: 10.1038/s41586-025-09422-z.  
645 URL <https://doi.org/10.1038/s41586-025-09422-z>.

646 Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song,  
647 and Jacob Steinhardt. Measuring mathematical problem solving with the MATH dataset. In

648        *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*  
 649        (*Round 2*), 2021. URL <https://openreview.net/forum?id=7Bywt2mQsCe>.  
 650

651        Or Honovich, Thomas Scialom, Omer Levy, and Timo Schick. Unnatural instructions: Tuning  
 652        language models with (almost) no human labor. In Anna Rogers, Jordan Boyd-Graber, and  
 653        Naoaki Okazaki (eds.), *Proceedings of the 61st Annual Meeting of the Association for Com-  
 654        putational Linguistics (Volume 1: Long Papers)*, pp. 14409–14428, Toronto, Canada, July  
 655        2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-long.806. URL  
 656        <https://aclanthology.org/2023.acl-long.806/>.

657        Hanxu Hu, Simon Yu, Pinzhen Chen, and Edoardo Ponti. Fine-tuning large language mod-  
 658        els with sequential instructions. In Luis Chiruzzo, Alan Ritter, and Lu Wang (eds.), *Pro-  
 659        ceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association  
 660        for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*,  
 661        pp. 5589–5610, Albuquerque, New Mexico, April 2025. Association for Computational Lin-  
 662        guistics. ISBN 979-8-89176-189-6. doi: 10.18653/v1/2025.naacl-long.288. URL <https://aclanthology.org/2025.naacl-long.288/>.

664        Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando  
 665        Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free eval-  
 666        uation of large language models for code. In *The Thirteenth International Conference on Learning  
 667        Representations*, 2025. URL <https://openreview.net/forum?id=chfJJYC3iL>.  
 668

669        Fangkai Jiao, Geyang Guo, Xingxing Zhang, Nancy F. Chen, Shafiq Joty, and Furu Wei. Prefer-  
 670        ence optimization for reasoning with pseudo feedback. In *The Thirteenth International Confer-  
 671        ence on Learning Representations*, 2025. URL <https://openreview.net/forum?id=jkUp3lybXf>.  
 672

673        Simran Kaur, Simon Park, Anirudh Goyal, and Sanjeev Arora. Instruct-skillmix: A powerful pipeline  
 674        for llm instruction tuning. In *The Thirteenth International Conference on Learning Representations*,  
 675        2025.  
 676

677        Bespoke Labs. Bespoke-stratos: The unreasonable effectiveness of reasoning distilla-  
 678        tion. [www.bespokelabs.ai/blog/bespoke-stratos-the-unreasonable-effectiveness-of-reasoning-  
 679        distillation](http://www.bespokelabs.ai/blog/bespoke-stratos-the-unreasonable-effectiveness-of-reasoning-distillation), 2025. Accessed: 2025-01-22.  
 680

681        Nathan Lambert, Jacob Morrison, Valentina Pyatkin, Shengyi Huang, Hamish Ivison, Faeze Brahman,  
 682        Lester James Validad Miranda, Alisa Liu, Nouha Dziri, Xinxi Lyu, Yuling Gu, Saumya Malik,  
 683        Victoria Graf, Jena D. Hwang, Jiangjiang Yang, Ronan Le Bras, Oyvind Tafjord, Christopher  
 684        Wilhelm, Luca Soldaini, Noah A. Smith, Yizhong Wang, Pradeep Dasigi, and Hannaneh Hajishirzi.  
 685        Tulu 3: Pushing frontiers in open language model post-training. In *Second Conference on Language  
 686        Modeling*, 2025. URL <https://openreview.net/forum?id=i1uGbfHHpH>.

687        Chen Li, Weiqi Wang, Jingcheng Hu, Yixuan Wei, Nanning Zheng, Han Hu, Zheng Zhang, and  
 688        Houwen Peng. Common 7b language models already possess strong math capabilities, 2024a.  
 689        URL <https://arxiv.org/abs/2403.04706>.  
 690

691        Haoran Li, Qingxiu Dong, Zhengyang Tang, Chaojun Wang, Xingxing Zhang, Haoyang Huang,  
 692        Shaohan Huang, Xiaolong Huang, Zeqiang Huang, Dongdong Zhang, Yuxian Gu, Xin Cheng,  
 693        Xun Wang, Si-Qing Chen, Li Dong, Wei Lu, Zhifang Sui, Benyou Wang, Wai Lam, and Furu Wei.  
 694        Synthetic data (almost) from scratch: Generalized instruction tuning for language models, 2024b.  
 695        URL <https://arxiv.org/abs/2402.13064>.

696        Junlong Li, Daya Guo, Dejian Yang, Runxin Xu, Yu Wu, and Junxian He. CodeIO: Condensing  
 697        reasoning patterns via code input-output prediction. In *Forty-second International Conference on  
 698        Machine Learning*, 2025. URL <https://openreview.net/forum?id=feIaF6vYf1>.  
 699

700        Rongao Li, Jie Fu, Bo-Wen Zhang, Tao Huang, Zhihong Sun, Chen Lyu, Guang Liu, Zhi Jin, and  
 701        Ge Li. Taco: Topics in algorithmic code generation dataset, 2023. URL <https://arxiv.org/abs/2312.14852>.

702 Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittweis, Rémi Leblond, Tom  
 703 Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien  
 704 de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven  
 705 Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson,  
 706 Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code  
 707 generation with alphacode. *Science*, 378(6624):1092–1097, December 2022. ISSN 1095-9203. doi:  
 708 10.1126/science.abq1158. URL <http://dx.doi.org/10.1126/science.abq1158>.

709  
 710 Xiao Liang, Zhong-Zhi Li, Yeyun Gong, Yang Wang, Hengyuan Zhang, Yelong Shen, Ying Nian Wu,  
 711 and Weizhu Chen. SwS: Self-aware weakness-driven problem synthesis in reinforcement learning  
 712 for llm reasoning, 2025. URL <https://arxiv.org/abs/2506.08989>.

713 Haipeng Luo, Qingfeng Sun, Can Xu, Pu Zhao, Jian-Guang Lou, Chongyang Tao, Xiubo Geng,  
 714 Qingwei Lin, Shifeng Chen, Yansong Tang, and Dongmei Zhang. Wizardmath: Empowering  
 715 mathematical reasoning for large language models via reinforced evol-instruct. In *The Thirteenth  
 716 International Conference on Learning Representations*, 2025a. URL <https://openreview.net/forum?id=mMPMHWOdOy>.

717  
 718 Michael Luo, Sijun Tan, Roy Huang, Ameen Patel, Alpay Ariyak, Qingyang  
 719 Wu, Xiaoxiang Shi, Rachel Xin, Colin Cai, Maurice Weber, Ce Zhang, Li Er-  
 720 ran Li, Raluca Ada Popa, and Ion Stoica. Deepcoder: A fully open-source  
 721 14b coder at o3-mini level. <https://pretty-radio-b75.notion.site/DeepCoder-A-Fully-Open-Source-14B-Coder-at-O3-mini-Level-1cf81902c14680b3bee5eb349a512a51>, 2025b. Notion Blog.

722  
 723  
 724 Niklas Muennighoff, Zitong Yang, Weijia Shi, Xiang Lisa Li, Li Fei-Fei, Hannaneh Hajishirzi, Luke  
 725 Zettlemoyer, Percy Liang, Emmanuel Candès, and Tatsunori Hashimoto. s1: Simple test-time  
 726 scaling, 2025. URL <https://arxiv.org/abs/2501.19393>.

727  
 728 Nvidia, Bo Adler, Niket Agarwal, Ashwath Aithal, Dong H. Anh, Pallab Bhattacharya, Annika Brun-  
 729 dyn, Jared Casper, Bryan Catanzaro, Sharon Clay, Jonathan Cohen, Sirshak Das, Ayush Dattagupta,  
 730 Olivier Delalleau, Leon Derczynski, Yi Dong, Daniel Egert, Ellie Evans, Aleksander Ficek, Denys  
 731 Fridman, Shaona Ghosh, Boris Ginsburg, Igor Gitman, Tomasz Grzegorzek, Robert Hero, Jining  
 732 Huang, Vibhu Jawa, Joseph Jennings, Aastha Jhunjhunwala, John Kamalu, Sadaf Khan, Oleksii  
 733 Kuchaiev, Patrick LeGresley, Hui Li, Jiwei Liu, Zihan Liu, Eileen Long, Ameya Sunil Mahabalesh-  
 734 warkar, Somshubra Majumdar, James Maki, Miguel Martinez, Maer Rodrigues de Melo, Ivan  
 735 Moshkov, Deepak Narayanan, Sean Narenthiran, Jesus Navarro, Phong Nguyen, Osvald Nitski,  
 736 Vahid Noroozi, Guruprasad Nutheti, Christopher Parisien, Jupinder Parmar, Mostofa Patwary,  
 737 Krzysztof Pawelec, Wei Ping, Shrimai Prabhumoye, Rajarshi Roy, Trisha Saar, Vasanth Rao Naik  
 738 Sabavat, Sanjeev Satheesh, Jane Polak Scowcroft, Jason Sewall, Pavel Shamis, Gerald Shen,  
 739 Mohammad Shoeybi, Dave Sizer, Misha Smelyanskiy, Felipe Soares, Makesh Narsimhan Sreedhar,  
 740 Dan Su, Sandeep Subramanian, Shengyang Sun, Shubham Toshniwal, Hao Wang, Zhilin Wang,  
 741 Jiaxuan You, Jiaqi Zeng, Jimmy Zhang, Jing Zhang, Vivienne Zhang, Yian Zhang, and Chen Zhu.  
 742 Nemotron-4 340b technical report, 2024. URL <https://arxiv.org/abs/2406.11704>.

743  
 744 OpenAI. Openai o1 system card, 2024. URL <https://arxiv.org/abs/2412.16720>.

745  
 746 Zeyu Qin, Qingxiu Dong, Xingxing Zhang, Li Dong, Xiaolong Huang, Ziyi Yang, MAHMOUD  
 747 KHADEMI, Dongdong Zhang, Hany Hassan Awadalla, Yi R. Fung, Weizhu Chen, Minhao Cheng,  
 748 and Furu Wei. Scaling laws of synthetic data for language model. In *Second Conference on  
 Language Modeling*, 2025. URL <https://openreview.net/forum?id=UmUXPXHtdl>.

749  
 750 Vedant Shah, Dingli Yu, Kaifeng Lyu, Simon Park, Jiatong Yu, Yinghui He, Nan Rosemary Ke,  
 751 Michael Mozer, Yoshua Bengio, Sanjeev Arora, and Anirudh Goyal. Ai-assisted generation of  
 752 difficult math questions, 2025. URL <https://arxiv.org/abs/2407.21009>.

753  
 754 Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang,  
 755 Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of  
 mathematical reasoning in open language models, 2024. URL <https://arxiv.org/abs/2402.03300>.

756 Ben Shi, Michael Tang, Karthik R Narasimhan, and Shunyu Yao. Can language models solve  
 757 olympiad programming? In *First Conference on Language Modeling*, 2024. URL <https://openreview.net/forum?id=kGa4fMtP91>.  
 758

760 Zhengyang Tang, Xingxing Zhang, Benyou Wang, and Furu Wei. Mathscales: Scaling instruction  
 761 tuning for mathematical reasoning. In *Forty-first International Conference on Machine Learning*,  
 762 2024. URL <https://openreview.net/forum?id=Kjww7ZN47M>.  
 763

764 Yuxuan Tong, Xiwen Zhang, Ruidong Wu, and Junxian He. DART-math: Difficulty-aware  
 765 rejection tuning for mathematical problem-solving. In *The Thirty-eighth Annual Conference on*  
 766 *Neural Information Processing Systems*, 2024. URL <https://openreview.net/forum?id=zLU21oQjD5>.  
 767

768 Shubham Toshniwal, Wei Du, Ivan Moshkov, Branislav Kisacanin, Alexan Ayrapetyan, and Igor  
 769 Gitman. Openmathinstruct-2: Accelerating AI for math with massive open-source instruction  
 770 data. In *The Thirteenth International Conference on Learning Representations*, 2025. URL  
 771 <https://openreview.net/forum?id=mTCbq2QssD>.  
 772

773 Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H. Chi, Sharan Narang, Aakanksha  
 774 Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language  
 775 models. In *The Eleventh International Conference on Learning Representations*, 2023a. URL  
 776 <https://openreview.net/forum?id=1PL1NIMMrw>.  
 777

778 Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and  
 779 Hannaneh Hajishirzi. Self-instruct: Aligning language models with self-generated instructions. In  
 780 Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki (eds.), *Proceedings of the 61st Annual*  
 781 *Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 13484–  
 782 13508, Toronto, Canada, July 2023b. Association for Computational Linguistics. doi: 10.18653/  
 783 v1/2023.acl-long.754. URL [https://aclanthology.org/2023.acl-long.754/](https://aclanthology.org/2023.acl-long.754).  
 784

785 Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed H. Chi, Quoc V  
 786 Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language models.  
 787 In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho (eds.), *Advances in*  
 788 *Neural Information Processing Systems*, 2022. URL [https://openreview.net/forum?id=\\_VjQlMeSB\\_J](https://openreview.net/forum?id=_VjQlMeSB_J).  
 789

790 Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, Qingwei  
 791 Lin, and Dixin Jiang. WizardLM: Empowering large pre-trained language models to follow  
 792 complex instructions. In *The Twelfth International Conference on Learning Representations*, 2024.  
 793 URL <https://openreview.net/forum?id=CfXh93NDgH>.  
 794

795 Zhangchen Xu, Fengqing Jiang, Luyao Niu, Yuntian Deng, Radha Poovendran, Yejin Choi, and  
 796 Bill Yuchen Lin. Magpie: Alignment data synthesis from scratch by prompting aligned LLMs with  
 797 nothing. In *The Thirteenth International Conference on Learning Representations*, 2025. URL  
 798 <https://openreview.net/forum?id=Pnk7vMboznK>.  
 799

800 Yixin Ye, Zhen Huang, Yang Xiao, Ethan Chern, Shijie Xia, and Pengfei Liu. LIMO: Less is more for  
 801 reasoning. In *Second Conference on Language Modeling*, 2025. URL <https://openreview.net/forum?id=T2TZ0RY4Zk>.  
 802

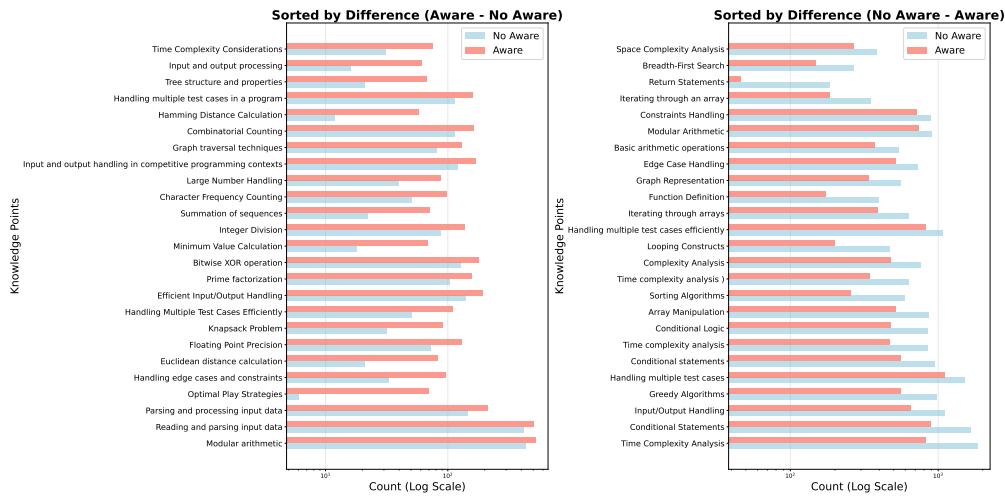
803 Longhui Yu, Weisen Jiang, Han Shi, Jincheng YU, Zhengying Liu, Yu Zhang, James Kwok, Zhenguo  
 804 Li, Adrian Weller, and Weiyang Liu. Metamath: Bootstrap your own mathematical questions for  
 805 large language models. In *The Twelfth International Conference on Learning Representations*,  
 806 2024. URL <https://openreview.net/forum?id=N8N0hgNDrt>.  
 807

808 Weizhe Yuan, Jane Yu, Song Jiang, Karthik Padthe, Yang Li, Ilia Kulikov, Kyunghyun Cho, Dong  
 809 Wang, Yuandong Tian, Jason E Weston, and Xian Li. NaturalReasoning: Reasoning in the wild  
 810 with 2.8m challenging questions, 2025. URL <https://arxiv.org/abs/2502.13124>.  
 811

812 Xiang Yue, Tuney Zheng, Ge Zhang, and Wenhui Chen. Mammoth2: Scaling instructions from the  
 813 web. *Advances in Neural Information Processing Systems*, 2024.

810 Xueliang Zhao, Wei Wu, Jian Guan, and Lingpeng Kong. PromptCoT: Synthesizing olympiad-  
811 level problems for mathematical reasoning in large language models. In Wanxiang Che, Joyce  
812 Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Findings of the Association for*  
813 *Computational Linguistics: ACL 2025*, pp. 18167–18188, Vienna, Austria, July 2025. Association  
814 for Computational Linguistics. ISBN 979-8-89176-256-5. doi: 10.18653/v1/2025.findings-acl.935.  
815 URL <https://aclanthology.org/2025.findings-acl.935/>.

816  
817  
818  
819  
820  
821  
822  
823  
824  
825  
826  
827  
828  
829  
830  
831  
832  
833  
834  
835  
836  
837  
838  
839  
840  
841  
842  
843  
844  
845  
846  
847  
848  
849  
850  
851  
852  
853  
854  
855  
856  
857  
858  
859  
860  
861  
862  
863

864 **A APPENDIX**  
865  
866  
867884 Figure 4: Examples of knowledge points where the count is increased (left) or decreased (right) when  
885 using difficulty-aware sampling.  
886  
887888 **A.1 USAGE OF LLM**  
889  
890 We used LLMs to polish the writing and assist in the implementation of parts of the codebase. All the  
891 methodologies, ideas, conceptual contributions, experiment designs, analyses were developed solely  
892 by the authors.  
893  
894  
895  
896  
897  
898  
899  
900  
901  
902  
903  
904  
905  
906  
907  
908  
909  
910  
911  
912  
913  
914  
915  
916  
917

918  
919  
920  
921  
922  
923  
924  
925  
926  
927  
928  
929  
930  
931

Table 6: Case Study: Topic and knowledge points that were provided to the generator, and two generated questions. Left is generated by our trained model, right is generated by original Qwen2.5-14B-Instruct.

| Topic            | String Analysis  |
|------------------|--|
| Knowledge points | 1. Hamming Distance Calculation; 2. Time Complexity Optimization; 3. Conditional Logic and State Management; 4. Event Handling and Simulation; 5. Priority Queue   |
| Problem          | <p>You are given a string <math>s</math> of length <math>n</math> consisting of binary characters ('0' and '1'). You need to perform several queries on this string. Each query can be one of the following types:</p> <ol style="list-style-type: none"> <li>1. <math>\text{set}(l, r, c)</math>: Set all characters in the substring <math>s[l:r+1]</math> to the character <math>c</math> (either '0' or '1').</li> <li>2. <math>\text{flip}(l, r)</math>: Flip all characters in the substring <math>s[l:r+1]</math> (change '0' to '1' and '1' to '0').</li> <li>3. <math>\text{hamming}(l1, r1, l2, r2)</math>: Calculate the Hamming distance between the substrings <math>s[l1:r1+1]</math> and <math>s[l2:r2+1]</math>.</li> </ol> <p>Your task is to handle these queries efficiently.</p> <p><b>Input:</b></p> <ul style="list-style-type: none"> <li>- The first line contains the string <math>s</math> of length <math>n</math> (<math>1 \leq n \leq 200,000</math>).</li> <li>- The second line contains an integer <math>q</math> (<math>1 \leq q \leq 400,000</math>) representing the number of queries.</li> <li>- The next <math>q</math> lines describe the queries. Each query is in one of the above formats.</li> <li>- For each hamming query, output the Hamming distance between the specified substrings.</li> </ul> |

961  
962  
963  
964  
965  
966  
967  
968  
969  
970  
971

In a messaging system, messages are sent as strings of binary digits. Sometimes, due to technical issues, certain bits in the messages might get flipped (0 becomes 1, and 1 becomes 0). To ensure the integrity of the messages, a special protocol is used where each message is sent alongside its "flipped" version. Upon receiving both versions, the receiver calculates the Hamming distance between the original and the flipped message to detect any errors.

Given two strings  $s$  and  $t$  of equal length representing the original and flipped versions of a message, your task is to write a function that calculates the Hamming distance between them. The Hamming distance is defined as the number of positions at which the corresponding bits are different. Additionally, the function should handle up to  $10^5$  queries efficiently, optimizing for time complexity.

```

972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025

    Act as a Programming Teacher and create a new question and its solution based on
    the provided topics and knowledge points. Ensure that the created questions:
    1. Adhere to the provided topics.
    2. Necessitate the combined use of the associated knowledge points.

    # Example
    Topics:
    1. Graph Theory
    2. Combinatorics
    Knowledge Points:
    1. Graph Coloring
    2. Connected Components
    3. Dynamic Programming
    4. Modular Arithmetic
    Created Question:
    You are given a grid, consisting of  $2n$  rows and  $n$  columns. Each cell of this grid
    should be colored either black or white.
    Two cells are considered neighbours if they have a common border and share the
    same color. Two cells  $A$  and  $B$  belong to the same component if they are
    neighbours, or if there is a neighbour of  $A$  that belongs to the same component with
     $B$ .
    Let's call some bicoloring beautiful if it has exactly  $k$  components.
    Count the number of beautiful bicolorings. The number can be big enough, so print the
    answer modulo  $998244353$ .
    ----Input----
    The only line contains two integers  $n$  and  $k$  ( $1 \leq n \leq 1000$ ,  $1 \leq k \leq 2n$ ) —
    the number of columns in a grid and the number of components required.
    ----Output---
    Print a single integer — the number of beautiful bicolorings modulo  $998244353$ .
    ----Examples----
    Input
    3 4
    Output
    12
    Input
    4 1
    Output
    2
    Input
    1 2
    Output
    2
    Topics:
    1. String Manipulation
    Knowledge Points:
    1. Understanding and manipulating string data structures
    2. Dynamic Programming

    Try to create a question for the last one. Structure your response as:
    Created Question:
    <Question>

```

Figure 5: 1-shot prompt example for problem generation. It is simplified for visualization, in real prompt, we have 8-shot for in-context learning.

```

1026
1027
1028 You are an expert programmer. Your task is to write some test cases to the programming problems to help verify the expected program solutions. You
1029 only need to give me the inputs in the required format. Now, let me introduce the details to you:
1030     ## Program Format
1031
1032     You will be given programming problems that accept standard input-output stream. As a result, the test case inputs should contain only the inputs text
1033     stream.
1034
1035     ## Response Format
1036
1037     You should return me the test case inputs in `json_object` format. You need to generate **20** groups of test case inputs, and each key field is named
1038     as `test_case_i`, where `i` is the index of the test case. The value of each key is the test case inputs in the required format.
1039
1040     ## Example for Standard Input-Output Stream
1041
1042     ##### Programming Problem
1043
1044     Polycarp has  $n$  different binary words. A word called binary if it contains only characters '0' and '1'. For example, these words are binary: "0001", "11",
1045     "0" and "001100".
1046
1047     Polycarp wants to offer his set of  $n$  binary words to play a game "words". In this game, players name words and each next word (starting from the
1048     second) must start with the last character of the previous word. The first word can be any. For example, these sequence of words can be named during
1049     the game: "0101", "1", "10", "00", "00001".
1050
1051     Word reversal is the operation of reversing the order of the characters. For example, the word "0111" after the reversal becomes "1110", the word
1052     "11010" after the reversal becomes "01011".
1053
1054     Probably, Polycarp has such a set of words that there is no way to put them in the order correspondent to the game rules. In this situation, he wants to
1055     reverse some words from his set so that: the final set of  $n$  words still contains different words (i.e. all words are unique); there is a way to put all
1056     words of the final set of words in the order so that the final sequence of  $n$  words is consistent with the game rules.
1057
1058     Polycarp wants to reverse minimal number of words. Please, help him.
1059
1060     -----Input-----
1061
1062     The first line of the input contains one integer  $k$  ( $1 \leq k \leq 10^4$ ) — the number of test cases in the input. Then  $k$  test cases follow.
1063
1064     The first line of a test case contains one integer  $n$  ( $1 \leq n \leq 2 \cdot 10^5$ ) — the number of words in the Polycarp's set. Next  $n$  lines contain these
1065     words. All of  $n$  words aren't empty and contains only characters '0' and '1'. The sum of word lengths doesn't exceed  $4 \cdot 10^6$ . All words are
1066     different.
1067
1068     Guaranteed, that the sum of  $n$  for all test cases in the input doesn't exceed  $2 \cdot 10^5$ . Also, guaranteed that the sum of word lengths for all test
1069     cases in the input doesn't exceed  $4 \cdot 10^6$ .
1070
1071     -----Output-----
1072
1073     Print answer for all of  $k$  test cases in the order they appear.
1074
1075     If there is no answer for the test case, print -1. Otherwise, the first line of the output should contain  $k$  ( $0 \leq k \leq n$ ) — the minimal number of words
1076     in the set which should be reversed. The second line of the output should contain  $k$  distinct integers — the indexes of the words in the set which
1077     should be reversed. Words are numerated from  $1$  to  $n$  in the order they appear. If  $k=0$  you can skip this line (or you can print an empty line). If
1078     there are many answers you can print any of them.
1079
1080     -----Example-----
1081
1082     Input
1083     4
1084     4
1085     0001
1086     1000
1087     0011
1088     0111
1089     3
1090     010
1091     101
1092     0
1093     2
1094     00000
1095     00001
1096     4
1097     01
1098     001
1099     0001
1100     00001
1101
1102     Output
1103     1
1104     3
1105     -1
1106     0
1107
1108     2
1109     1 2
1110
1111     ##### Response
1112
1113     {
1114         "test_case_0": "3\n3\n101\n110\n011\n10\n01\n10\n4\n0001\n1000\n0011\n0111",
1115         "test_case_1": "2\n2\n01\n10\n3\n000\n111\n110",
1116         ...
1117     }
1118
1119     ## Get Started
1120
1121     Note that in the above example, I omit some test case inputs. You should return **20** groups of inputs to me in `json_object` format.
1122
1123     ##### Programming Problem
1124
1125     {problem}
1126
1127     ##### Response

```

Figure 6: 1-shot example prompt for testcase generation.

```

1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098 Act as a Programming Teacher and analyze the provided question. Start by identifying 1 or 2 general topics
1099 that a student is being assessed on. Structure your response as:
1100 "Topics:
1101 1. <Topic 1>
1102 2. <Topic 2>"
```

1103 Next, highlight 1 to 5 specific knowledge points that the question evaluates. Structure your response as:
1104 "Specific Knowledge Points:
1105 1. <Knowledge Point 1>
1106 2. <Knowledge Point 2>
1107 3. <Knowledge Point 3>
1108 4. <Knowledge Point 4>
1109 5. <Knowledge Point 5>"

1110 The topics and specific knowledge points should be terms that are concise and commonly used in academia
1111 or industry.

1112 ### Provided question:  
1113 {{ question }}

1114 ### Analysis:

Figure 7: Prompt demonstration for concept extraction.